

CS 161: Example Homework Solution

To give you a sense for what a good homework solution might look like, we've provided these sample problems along with solutions that would earn full credit. We suggest taking some time to work through these problems so that you have a sense for how to solve them. You should have the prerequisite knowledge to do this after the end of week 1.

Problem: Sorting a k -somewhat-sorted array. You are given an array A of distinct integers for which it's possible to remove k elements from A and obtain a sorted array. Design an $O(n)$ -time algorithm for determining $O(k)$ elements that can be removed from A , resulting in a sorted array. Your algorithm can use $O(k)$ additional space. Convince a colleague of the correctness, runtime, and space usage of your algorithm.

Solution. Our algorithm will run as follows: initialize $A_{prev} = A[0]$ and $A_{cur} = A[1]$. Iterate through the array and repeat the following steps:

- Mark elements A_{prev} and A_{cur} as “discarded” if $A_{prev} > A_{cur}$.
- If the elements are not discarded, push A_{prev} to a deque (double-ended queue with $O(1)$ `pop_top`, `pop_bottom`, and `push_top`) and set $A_{prev} = A_{cur}$ and $A_{cur} = A_{cur+1}$ (the next element in A). If the size of the deque exceeds k , pop from the bottom of the deque.
- If the elements are discarded, pop from the top of the deque and use this value as A_{prev} , and set $A_{cur} = A_{cur+1}$.

Correctness. To prove correctness, we will show that the following is always true: (*) The undiscarded elements of the subarray $A_0, \dots, A_{cur-1}$ form a sorted array. This statement is true when the algorithm begins, since an array with one element is trivially sorted. At each iteration, we consider two cases:

1. $A_{prev} > A_{cur}$. Since (*) holds on entry to the loop, we know that after discarding A_{prev} and A_{cur} , (*) will still hold for all undiscarded elements.
2. $A_{prev} < A_{cur}$. If (*) holds on entry to the loop, then A_{prev} is greater than all other undiscarded values in the subarray. Since $A_{prev} < A_{cur}$, then A_{cur} must be greater than all other undiscarded values in the subarray, including A_{prev} . Thus, the undiscarded elements of $A_0, \dots, A_{cur-1}, A_{cur}$ must form a sorted array, and (*) still holds.

Using the fact that (*) is true, we now argue correctness. When the algorithm concludes, $0, \dots, A_{cur}$ encompasses all of A , so all undiscarded elements from A form a sorted array, as desired. In addition, the algorithm enters case (a) no more than k times for a k -somewhat-sorted array. This is because each time we enter case (a), it must be because of two elements which are out of order compared to each other, which can only happen at most k times in a k -somewhat-sorted array. Thus, the algorithm pops from the top of the stack at most k times and marks at most $2k = O(k)$ elements to be discarded, as desired.

Runtime. This algorithm runs in $O(n)$. To see this, note that each iteration takes $O(1)$ work, then increments cur by one. Since the algorithm starts with $cur = 1$ and terminates when cur reaches the end of A , this means that the algorithm runs for $O(n)$ iterations. Since each iteration does $O(1)$ work, the total runtime is $O(n)$.

Space usage. Since the algorithm maintains a deque of size at most $O(k)$, it uses $O(k)$ extra space.